

15 Objekte, Klassen und Methoden

*I invented the term Object-Oriented,
and I can tell you I did not have C++ in mind.*
Alan Kay



Ein Ruby-Programm besteht aus Botschaften, die zwischen Objekten hin und her geschickt werden. Dieses Verfahren ermöglicht einen sehr »natürlichen« Umgang mit Objekten, den wir in diesem Kapitel näher beschreiben wollen. Dazu beginnen wir mit einer kurzen, Ruby-orientierten Einführung in objektorientiertes Programmieren (OOP), um verwendete Begriffe und Modellvorstellungen zu klären. Hauptsächlich soll dieses Kapitel aber zeigen, wie man in Ruby mit Klassen, Objekten und Methoden (Implementation von Botschaften) umgeht.

15.1 Objektorientierte Programmierung

Die folgenden kurzen Abschnitte sollen die in diesem Buch verwendeten Begriffe klären beziehungsweise als Auffrischung dienen. Wer noch nie mit objektorientierten Programmiersprachen gearbeitet hat, sollte sie mindestens zwei Mal lesen, um heuristische Effekte auszunutzen und sich mit einer ausführlichen Anleitung zu diesem Thema etwas Gutes tun. Ein sehr gute Einführung in OOP ist das auf Smalltalk ausgerichtete Buch [Liu96], dessen Inhalte wegen der engen Verwandtschaft der zu Grunde liegenden Modelle sehr leicht auf Ruby-Verhältnisse übertragen werden können.

15.1.1 Grundbegriffe

Ein *Objekt* kombiniert Wissen und Fähigkeiten. Das »Wissen« eines Objekts wird in seinen so genannten Instanzvariablen abgelegt. Diese Variablen sind von außerhalb des Objekts nicht zugänglich.

Objekte interagieren, indem sie sich *Botschaften* (Messages) zusenden. Wie der Empfänger auf die Botschaft reagiert, ist seine Sache. Im Normalfall wird eine *Methode* aufgerufen, in der das gewünschte Verhalten implementiert ist.

Klassen erzeugen Objekte und speichern die Implementationen der Methoden, die von den Objekten unterstützt werden. Daneben speichern Klassen noch Daten, die nicht von einzelnen Objekten abhängen, in so genannten Klassenvariablen oder Klasseninstanzvariablen. Auch diese sind von außen nicht direkt zugänglich.

Module sind in Ruby Klassen, die keine Objekte erzeugen können und nicht wie Klassen in einer Hierarchie angeordnet sind. Module können andere Module mit `include` einschließen, aber nicht voneinander abgeleitet werden. Sie werden in Kapitel 16 näher besprochen.

Klassen können voneinander abgeleitet werden. Dabei *vererbt* die vorhandene Klasse Wissen und Fähigkeiten an ihren Abkömmling. In der abgeleiteten Klasse müssen dann nur noch die Änderungen implementiert werden, die sie von ihrem Vorfahren unterscheiden.

Jedes Objekt in Ruby ist *Instanz* (Ausprägung) einer Klasse und enthält auch einen Verweis auf diese. Auch Klassen sind Objekte, nämlich Instanzen der Klasse `Class`. Alle Klassen sind (implizit und oft über mehrere Stufen) von der Klasse `Object` abgeleitet.

15.1.2 Prinzipien der OOP

Objektorientierte Programmierung beruht auf vier Grundlagen:

- ❑ **Abstraktion:** Hier geht es darum, die Gemeinsamkeiten von Objekten aufzuspüren, »natürliche« Gruppierungen zu bilden, in die sich die Objekte möglichst zwanglos einordnen. Vergleicht man die Ansprüche, die man an Arrays und Hash-Tabellen hat, findet man, dass beide Male die Möglichkeit benötigt wird, alle Elemente der Reihe nach abzuarbeiten, Elemente miteinander zu vergleichen oder Elemente mit einer bestimmten Eigenschaft herauszufiltern. Diese (und weitere) Fähigkeiten werden in Ruby im Modul `Enumerables` (=aufzählbare Dinge) zusammengefasst.
- ❑ **Kapselung:** Ein Objekt verbirgt interne Details vor der Außenwelt und kann so jederzeit umgestaltet werden, ohne von ihm abhängige Programmteile zu beeinflussen.

Ein Währungsobjekt wird beispielsweise eine Methode implementieren, um seinen Wert in Euro darzustellen. Wenn die Ursprungswährung schon Euro war, handelt es sich nur darum, den gespeicherten Zahlenwert auszugeben. Speichert das Objekt dagegen einen DM-Betrag, wird er vor der Ausgabe durch 1,95583 geteilt. Sollte es eine Währung mit einem freien Wechselkurs sein, könnte möglicherweise online der aktuelle Umrechnungsfaktor abgefragt werden, bevor eine Ausgabe stattfindet. In jedem Fall ist es Sache des Währungsobjekts, *seinen eigenen* Wert für die Ausgabe aufzubereiten.

- Polymorphismus: Ein und dieselbe Botschaft wird von unterschiedlichen Objekten unterschiedlich interpretiert. Anstatt dem Programmierer für die Darstellung grafischer Objekte eine Liste von Funktionen wie `drawRect`, `drawOval`, `drawArc`, `drawPolygon` etc. zu präsentieren, gibt es eine einzige Methode `draw`, die an Rechtecke, Ovale, Bögen oder Polygone geschickt wird.
- Vererbung: Wenn eine vorhandene Klasse schon fast das gewünschte Verhalten zeigt, ist es rationeller, nur die fehlenden Teile zu implementieren und für die übrigen sagen zu können: »Mach das so wie bisher auch.« Die neue Unterklasse ist damit eine vollständige Kopie der schon vorhandenen Oberklasse und unterscheidet sich nur in den explizit vorgenommenen Änderungen.

Neben der besseren Schreibökonomie hat man so auch einen wesentlichen Vorteil, wenn es darum geht, bestehenden Code zu ändern, denn die sauber durchgeführte Vererbung von Implementationen trägt dazu bei, dass der Code im Idealfall nur an einer Stelle angepasst werden muss, um für eine ganze Klassenhierarchie das Verhalten zu modifizieren.

15.1.3 Die Strategie der OOP

Man kann bei der objektorientierten Analyse und Lösung von Problemen vier Phasen unterscheiden:

1. Am Anfang steht eine ausführliche Beschäftigung mit der Problemstellung. Man wird Beispiele durchspielen, normale Situationen ebenso wie Grenzfälle, um sicherzustellen, dass man das Problem wirklich verstanden hat.

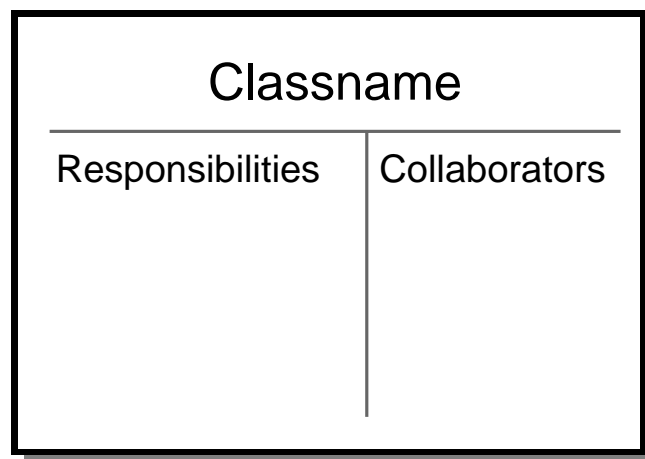
2. Als Nächstes werden die Objekte identifiziert, die sich in dem Problem verstecken. Man kann auf ihre Spur kommen, indem man versucht, die kleinen Probleme zu isolieren, deren Zusammenstellung das große Problem ausmacht.
3. Nun werden die Botschaften festgelegt, die zwischen den Akteuren ausgetauscht werden sollen. Auf einem gewissen Niveau werden hier Fachbegriffe des Problemfeldes vorkommen.
4. Schließlich werden die notwendigen Implementationen vorgenommen, wobei man stets darauf bedacht sein sollte, die zu Grunde liegende Sprache so viel wie möglich selbst erledigen zu lassen.

Die besten Ergebnisse erzielt man, wenn die Schritte nicht nur einmal abgearbeitet werden, sondern das Feedback aus späteren Phasen für eine weitere Iteration durch den Prozess verwendet wird. Mit zunehmender Übung verwischen sich auch die Grenzen zwischen diesen Phasen.

Eine sehr hilfreiche Technik für die Identifikation der Objekte und Botschaften sind die von Beck und Cunningham entwickelten CRC-Karten (Classname, Responsibility, Collaborators), deren Handhabung ebenfalls in [Liu96] ausführlich beschrieben ist.

CRC-Karten enthalten neben dem Namen der Klasse eine Auflistung der Fähigkeiten, über die ein Objekt verfügen sollte, sowie die Klassen, mit denen die aktuelle Klasse eng zusammenarbeitet.

Abbildung 15.1
Aufbau einer
CRC-Karte



Sie werden auf (preiswerten) Karteikarten angelegt und stellen eine sehr effektive Möglichkeit dar, die Interaktion der verschiedenen Objekte anzugeben.

15.2 Klassen und Methoden definieren

Die Definition einer Klasse erfolgt mit dem Schlüsselwort `class`:

```
class A
  def to_s; "ein A"; end
end
p A.new #-> "ein A"
```

Innerhalb einer Klasse werden mit `def` Methoden festgelegt, welche die von ihr erzeugten Instanzen verwenden können. Die Methode `to_s` liefert einen String, der das Objekt repräsentiert. Sie wird von Ruby verwendet, wenn eine schriftliche Repräsentation des Objekts erstellt werden muss.

Um eine neue Klasse zu erzeugen, die auf einer vorhandenen Klasse aufbaut, wird folgende Syntax verwendet:

```
class B < A
  def to_s
    super << "bkömmeling"
  end
  def to_i; 0; end
end

p B.new #-> "ein Abkömmling"
p B.new.to_i #-> 0
p A.new.to_i #-> NoMethodError
```



Übrigens kann man mit den Operatoren `<`, `>`, `<=` und `>=` überprüfen, ob eine Klasse von einer anderen abgeleitet ist:

```
B > A # B Oberklasse von A? #-> false
B < A # A Oberklasse von B? #-> true
```



Mit `super` wird die gleichnamige Methode der Oberklasse aufgerufen. Dieser Aufruf wird vor allem dann verwendet, wenn man in einer abgeleiteten Klasse zur in der Oberklasse vorhandenen Funktionalität etwas hinzufügen möchte. Methoden, die in der

Unterklasse hinzugefügt oder überschrieben werden, beeinflussen die Oberklasse nicht.

Um ein Objekt mit einer Instanzvariablen zu bestücken, reicht es, diese »irgendwann« zu verwenden:

```
class C
  def ivar; @ivar; end
  def ivar=(value)
    @ivar = value
  end
end
c = C.new
c.ivar      #-> nil
c.ivar = 5  #-> 5
c.ivar      #-> 5
```



Es kommt oft vor, dass man der Außenwelt Zugriff auf die Instanzvariablen eines Objekts gewähren will. Wenn es reine Lese- und Schreibmethoden sind, wie im oberen Beispiel, kann man auch folgendes (zu oben gleichwertiges) Verfahren verwenden:

```
class C2
  attr_reader :ivar
  attr_writer :ivar
end
#oder
class C3
  attr_accessor :ivar
end
```

Da Klassendefinitionen in Ruby ausführbarer Code sind, muss der Interpreter Konstruktionen wie die `attr`-Funktionen nicht gesondert behandeln. Sie funktionieren ähnlich wie Makros in Common Lisp und sind eigentlich Methoden, die das erzeugte `Class`-Objekt von seiner Oberklasse `Module` geerbt hat.

Anhand der Implementation der `attr_reader`-Methode sei erklärt, wie Ruby auf Grund seiner mächtigen Reflektionsmechanismen (siehe Kapitel 17) diese Aufgabe lösen kann.

```
class Module
  def attr_reader(*method_symbols)
    method_symbols.each { |method|
      module_eval(
```

```

        "def #{method}; return @#{method}; end")
    }
  end
end

```



Gelegentlich benötigt man Klassenvariablen, die einen instanz-unabhängigen Wert speichern. Solche Klassenvariablen müssen vor dem ersten Lesezugriff initialisiert werden. Im folgenden Beispiel lassen wir zählen, wie viele Instanzen einer Klasse schon erzeugt wurden. Die Methode `initialize` wird beim Erzeugen einer neuen Instanz automatisch aufgerufen.

```

class D
  @@nr = 0
  def initialize; @@nr += 1; end
  def D.status; puts @@nr; end
end
1.upto(rand 10) do D.new end
D.status

```

Neben Instanzmethoden, die sich an ein bestimmtes Objekt einer Klasse richten, gibt es noch Klassenmethoden, die eine Klasse als Empfänger haben. Nun sind aber Klassen in Ruby auch nur Instanzen der Klasse `Class`, so dass es keinen fundamentalen Unterschied zwischen Instanz- und Klassenmethoden gibt.

Um eine solche *Klassenmethode* zu definieren, wird in der `def`-Anweisung der Klassenname eingefügt. Im obigen Beispiel wird `D.status` unabhängig von Instanzen der Klasse aufgerufen.

15.3 Die Klassenhierarchie

Wir können Rubys Klassenhierarchie mit Bordmitteln erforschen: Mit `superclass` erhalten wir die Basisklasse, von der die aktuelle Klasse abgeleitet wurde. Die Methode `ancestors` zeigt uns alle Oberklassen und Module der angesprochenen Klasse, in der Reihenfolge, in der diese nach Methoden durchsucht werden.

Ruby erzählt von sich

```

def lineage(cls)
  puts "#{cls} -> #{cls.superclass}"
  p cls.ancestors.collect { |m|
    [m, m.class]
  }
}

```

```

end

lineage String

# Ausgabe:
String -> Object
[[String, Class], [Enumerable, Module],
 [Comparable, Module], [Object, Class],
 [Kernel, Module]]

```

Der Ausgabe entnehmen wir, dass die Klasse `String` direkt von `Object` abstammt, aber die Module `Enumerable` und `Comparable` eingebunden hat. Die Klasse `Object` wiederum verwendet das Modul `Kernel`. Mehr Informationen zu Modulen gibt es in Kapitel 16.

Wenn wir das Modul `ObjectSpace` verwenden, können wir auch »von oben« kommen und zum Beispiel alle Unterklassen von `Numeric` ausfindig machen:

```

def subclasses(cls)
  sub = []
  ObjectSpace.each_object{ |obj|
    if obj.class == Class and
      obj.ancestors.member? cls
      sub << [obj.ancestors.index(cls), obj]
    end
  }
  sub.sort! { |a, b| a[0] <=> b[0] }
  sub.each{ |i|
    puts "#{i[1]} -> #{i[1].superclass}"
  }
end

subclasses Numeric

# Ausgabe:
Numeric -> Object
Float -> Numeric
Integer -> Numeric
Fixnum -> Integer
Bignum -> Integer

```

15.4 Besonderheiten bei def

15.4.1 Parameterangaben

Wenn beim Methodenaufruf Argumente übergeben werden sollen, folgt die entsprechende Auflistung dem Funktionsnamen, wobei mehrere Parameter durch Komma getrennt werden. Das im Abschnitt 7.2 beschriebene Notationsschema findet auch hier Anwendung.

```
def ohne; 0; end
def mitEinem(a); a; end
def mitVielen(*a); a; end
def mitBlock(&a); a.class; end
def kombi(a, b, *c); p [a, b, c]; end

ohne           #-> 0
mitEinem 1     #-> 1
mitVielen 1, 2, 3 #-> [1, 2, 3]
mitBlock { 1 } #-> Proc
kombi 1, 2, 3, 4 #-> [1, 2, [3, 4]]
```

15.4.2 Rückgabewerte

Der in einer Methode zuletzt ausgewertete Ausdruck stellt normalerweise den Rückgabewert. Mit dem Schlüsselwort `return` kann man die Methode von einer beliebigen Stelle aus verlassen und gleichzeitig einen Wert oder auch mehrere Werte (in einem automatisch erzeugten Array) zurückgeben:

```
def ret(n)
  if n % 2 == 0
    return "gerade", 2, 4, 6
  else
    return "ungerade", "der Rest"
  end
end
42
end
ret(0) #-> "gerade", 2, 4, 6
ret(1) #-> "ungerade", "der Rest"
```

15.4.3 Sichtbarkeitsbegrenzer

Durch das Vorstellen von `public`, `private` oder `protected` kann man ähnlich wie in anderen Sprachen die Sichtbarkeit von Methoden regeln.

*public = überall
verfügbar*

- ❑ `public`: Die Methode ist überall verfügbar. Wenn in Klassendefinitionen nichts anderes angegeben wird, werden bis auf `initialize` alle Methoden `public`.
- ❑ `protected`: Die Methode kann nur für die Definition von Methoden der aktuellen Klasse und ihrer Unterklassen verwendet werden. Bei einem expliziten Aufruf erhält man einen `NameError`.

private = nur für self

- ❑ `private`: Der Aufruf ist nur mit dem »impliziten Empfänger« möglich. Damit ist gemeint, dass kein Empfänger vor der Methode steht, aber das aktuelle Objekt `self` gemeint ist, dessen Methode gerade ausgeführt wird. Bereits die explizite Angabe von `self` funktioniert nicht mehr. Also sind `private`-Methoden aus der definierenden Klasse und auch deren Unterklassen verfügbar, aber nur für das jeweils aktuelle Objekt. `initialize` ist immer `private`.

```
class A
  private
    def priv; puts "private"; end
  protected
    def prot; puts "protected"; end
  public
    def publ; puts "public"; end

  def a_priv; priv; end
  def a_prot; prot; end
  def a_publ; publ; end
  def a_fail; self.priv; end # expl. Empfänger
end

a = A.new
a.a_priv #-> private
a.a_prot #-> protected
a.a_publ #-> public
a.a_fail #-> NameError
```

```
a.priv      #-> NameError
a.prot      #-> NameError
a.publ      #-> public

class B < A
  def b_priv; priv; end  # priv ist private in A
  def b_prot; prot; end
  def b_publ; publ; end
end

b = B.new
b.b_priv    #-> private
b.b_prot    #-> protected
b.b_publ    #-> public
```

Die Zugriffsrechte werden in Ruby dynamisch vergeben und können während der Laufzeit verändert werden.

15.4.4 Objekte und Singleton-Methoden

Man kann in Ruby auch Methoden definieren, die nur für ein ganz bestimmtes Objekt (Singleton) gelten. Solche Methoden werden Singleton-Methoden genannt.

```
class Ding; def to_s; "ding"; end; end
a = Ding.new
b = a.clone
def a.to_s; "a"; end
a      #-> a
b      #-> ding

#Kontrolle
a.singleton_methods #-> ["to_s"]
b.singleton_methods #-> []
```

Wird allerdings die Reihenfolge vertauscht, ändert sich das Ergebnis:

```
class Ding; def to_s; "ding"; end; end
a = Ding.new
def a.to_s; "a"; end
b = a.clone
a      #-> a
b      #-> a
```

```
#Kontrolle
a.singleton_methods #-> ["to_s"]
b.singleton_methods #-> ["to_s"]
```



Nachdem Klassen auch nur Objekte sind, müssten Klassenmethoden eigentlich Singleton-Methoden der Klassenobjekte sein. In der Tat:

```
class Ding; def Ding.xxx; end; end
Ding.singleton_methods
["xxx"]
```



*class << obj = Neue
Klasse nur für Objekt
obj*

Sollte man mehr als eine Singleton-Methode für ein Objekt definieren wollen, kann man in Anlehnung an die Vererbung bei Klassen Folgendes schreiben:

```
s="Ruby"
class <<s
  def to_i; 4; end
  def to_f; 4.0; end
end
s.to_i #-> 4
s.to_f #-> 4.0
```



Vor allem in kleinen Skripten definiert man Methoden außerhalb einer Klasse. Diese Methoden werden private Methoden der Klasse Object und mit dem so genannten Toplevel-Objekt main als impliziten Empfänger aufgerufen, wie das folgende Beispiel zeigt:

```
def ungebunden; self; end
id #-> 538233304
self.id #-> 538233304
self #-> main
self.class #-> Object
Object.private_methods #-> [ungebunden, ...]
ungebunden #-> main
```



15.5 Ändern von Klassen

Bei der Definition einer Klasse muss man sich entscheiden, wie die neue Klasse mit dem Rest des Ruby-Systems integriert werden soll. Sollen andere Ruby-Objekte als Klassen- oder Instanzvariablen eingebettet werden (Komposition)? Soll die Klasse direkt von einer anderen Klasse abgeleitet werden (Vererbung)? Letzteren Fall kann man in der `LogHash`-Klasse in Kapitel 14 begutachten.

Bei der direkten Ableitung müssen nur die Methoden neu definiert werden, die eine Erweiterung oder Änderung des Verhaltens der vorhandenen Klasse erfordern. Um eine schon vorhandene Methode der Oberklasse aufzurufen, verwendet man `super` mit den entsprechenden Argumenten.

```
class Punkt3d
  attr_reader :x, :y, :z, :name
  @@typ = "Punkt"
  def initialize(name, x, y, z)
    @x = x
    @y = y
    @z = z
    @name = name
  end
  def to_s
    "%s(%6.2f/%6.2f/%6.2f)" % [@name, @x, @y, @z]
  end
end

class Punkt2d < Punkt3d
  def initialize(name, x, y)
    super(name, x, y, 0)
  end
  def to_s
    "%s(%6.2f/%6.2f)" % [@name, @x, @y]
  end
end

class Strecke
  attr_reader :a, :b, :name
  @@typ = "Strecke"
  def initialize(name, a, b)
    @name = name
    @a = a
  end
end
```

```

        @b = b
      end
      def to_s
        "%s[%s/%s]" % [@name, @a, @b]
      end
    end
  end

p = Punkt2d.new("P", 1, 1)
q = Punkt2d.new("Q", 2, 3)
puts s = Strecke.new("s", p, q)
#-> s[P( 1.00/ 1.00)/Q( 2.00/ 3.00)]

```

Das erneute Definieren einer Klasse wird als *inkrementelle Änderung* aufgefasst, so dass eigene Methoden jederzeit hinzugefügt, aber auch überschrieben – d. h. neu definiert – werden können.

Der erste Vergleich im folgenden Beispiel verwendet die von der Object-Klasse ererbte Methode `==`, mit der Objektidentität geprüft wird. Nach Erweiterung der Klasse `Punkt3d` wird eine geometrische Interpretation der Gleichheit verwendet, so dass gleiche Koordinaten für die Gleichheit der Punkte ausreichen.

```

puts p == Punkt2d.new("P", 1, 1) #-> false

class Punkt3d
  def == (p)
    x == p.x and y == p.y and z == p.z
  end
end

puts p == Punkt2d.new("P", 1, 1) #-> true

```

Im Gegensatz zu Java sind auch die eingebauten Ruby-Klassen nicht »final«, also unveränderbar, sondern können bei Bedarf angepasst werden.



Bei Bedarf kann man eine Klasse mit `freeze` fixieren, so dass keine Änderungen mehr vorgenommen werden können. Jedoch wird beim Duplizieren mit `dup` stets eine nicht eingefrorene Kopie erzeugt, so dass man immer eine modifizierbare Klasse erhalten kann.

```

class Foo
  def to_s; "Foo"; end
end
Foo.freeze

```

```
Foo.new    #-> Foo

# Änderung der Klasse ist nicht möglich
class Foo
  def to_s; "Bar"; end #-> TypeError
end

# die Kopie ist wieder wandelbar
MyFoo = Foo.dup
class MyFoo
  def to_s; "Bar"; end
end
MyFoo.new  #-> Bar
```



Erweiterungen sind auch bei eingebauten Klassen möglich. Es wäre praktisch, wenn ein Array zählen könnte, wie oft es ein bestimmtes Objekt enthält. Dafür muss man nicht viel unternehmen:

```
class Array
  def zaehle(x)
    (select { |i| i == x }).size
  end
end

a = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 2, 3]
a.zaehle 2    #-> 3
a.zaehle 88   #-> 0
```

15.6 Klassenvariablen und Vererbung

Eine Klassenvariable stellt einen Zugang zu einem bestimmten Objekt dar, das allen Instanzen *der Klasse und ihren Unterklassen* gemein ist. Wenn man diese (von Smalltalk übernommene) Eigenschaft außer Acht lässt, gerät man schnell in Verwirrung, wie das folgende Beispiel aus der Ruby-Mailingliste zeigt.

```
class Fahrzeug
  @@reifen = 0
  def reifen
```

```

        @@reifen
      end
    end

    class Auto < Fahrzeug
      @@reifen = 4
    end

    class Fahrrad < Fahrzeug
      @@reifen = 2
    end

    puts Auto.new.reifen #-> 2 ??? Warum ???

```

Eine Klassenvariable ist innerhalb ihrer definierenden Klasse *und deren Unterklassen* global, so dass in obigem Beispiel also genau einer Variablen nacheinander drei Werte zugewiesen worden sind.

Um dieses Problem zu lösen, gibt es verschiedene Wege: Wenn sich der gespeicherte Wert nicht ändert, kann man eine klassenspezifische Konstante verwenden:

```

class Fahrzeug
  def reifen; type::REIFEN; end
end
class Auto < Fahrzeug; REIFEN = 4; end
class Fahrrad < Fahrzeug; REIFEN = 2; end
puts Auto.new.reifen #-> 4

```

Möchte man klassenspezifische Werte speichern, die sich ändern können, bieten sich Klasseninstanzvariablen an: Ebenso wie jedes Objekt seine eigenen Instanzvariablen hat, so hat jede Klasse (Instanz der Klasse Class) ihre eigenen Klasseninstanzvariablen, die nicht mit anderen Klassen geteilt werden.

```

class Fahrzeug
  @reifen = 0
  def Fahrzeug.reifen # Klassenmethode
    @reifen
  end
  def Fahrzeug.reifen=(zahl)
    @reifen = zahl
  end
end

```

```
class Auto < Fahrzeug; @reifen = 4; end
class Fahrrad < Fahrzeug; @reifen = 2; end

puts Auto.new.reifen      #-> 4
puts Fahrrad.new.reifen  #-> 2
```



Eine von Guy Decoux vorgeschlagene Lösung ist ein sehr gutes Beispiel für Ruby-typische Programmier Techniken. Sie erlaubt es, die oben manuell definierten Methoden kompakt als »normale« Akzessoren zu beschreiben:

```
class Fahrzeug
  @reifen = 0
  class << self; attr_accessor :reifen; end
end
class Auto < Fahrzeug; @reifen = 4; end
class Fahrrad < Fahrzeug; @reifen = 2; end
puts Auto.reifen, Fahrrad.reifen  #-> 4 2
```

Hier werden innerhalb der Klassendefinition Singleton-Methoden für die Fahrzeugklasse erzeugt, die auf die jeweils lokal gespeicherte Klasseninstanzvariable zugreifen. 